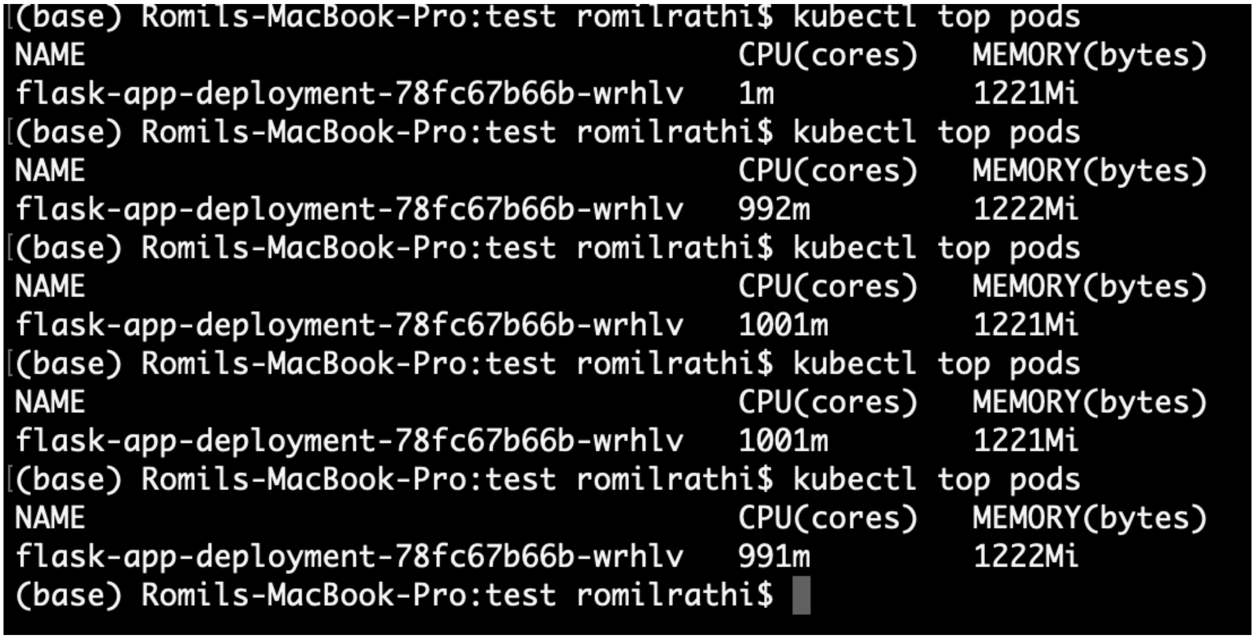
Load Balancer Test Results using Hey

Test for 50 QPS

The Load Balancer Test was conducted on a Minikube deployment with one replica using the following Hey command with minikube deployment for 1 replica.:

“hey -q 50 -z 5m -m POST -H "Content-Type: application/json" -d '{"query":"cordless drill", "topk": 10}' http://localhost:52519/predict”

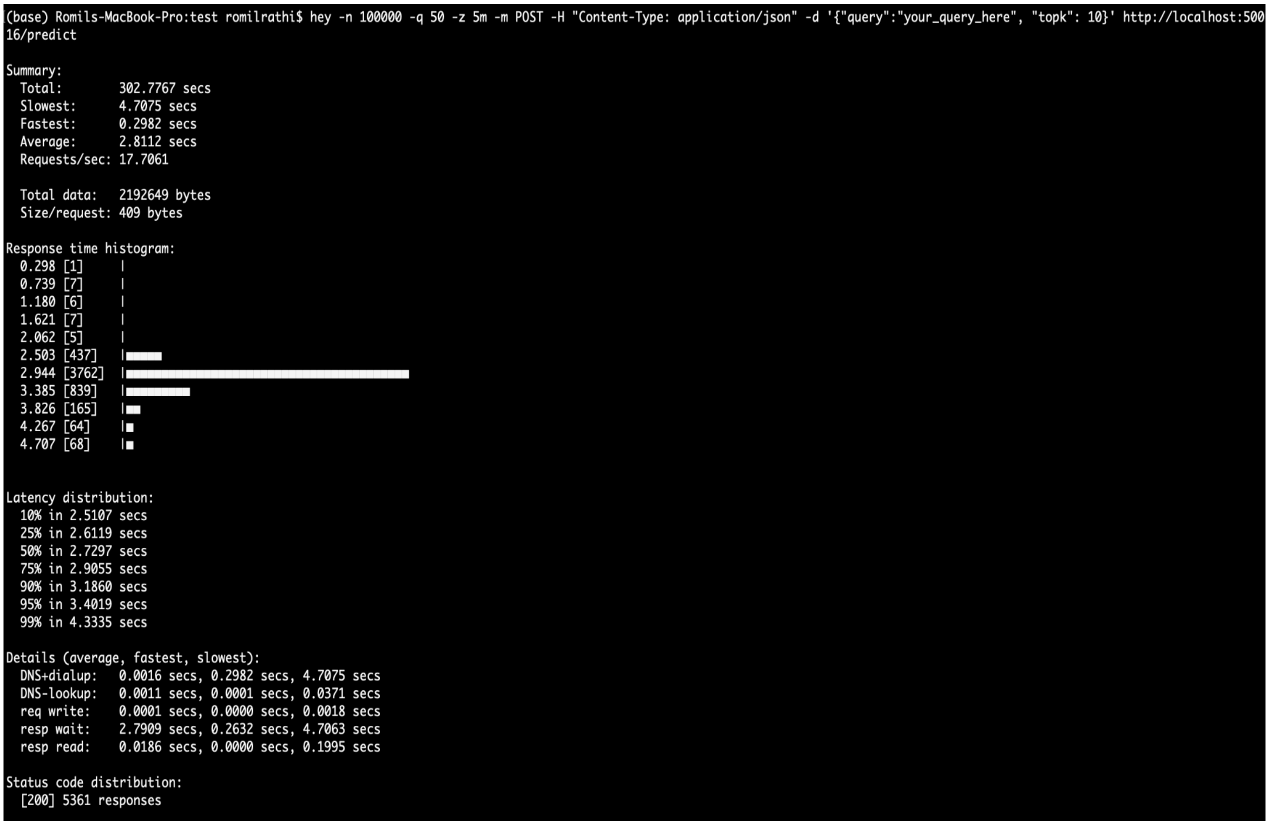
CPU Utilization Observations



```
[(base) Romils-MacBook-Pro:test romilrathi$ kubectl top pods
NAME                                       CPU(cores)   MEMORY(bytes)
flask-app-deployment-78fc67b66b-wrhlv   1m           1221Mi
[(base) Romils-MacBook-Pro:test romilrathi$ kubectl top pods
NAME                                       CPU(cores)   MEMORY(bytes)
flask-app-deployment-78fc67b66b-wrhlv   992m         1222Mi
[(base) Romils-MacBook-Pro:test romilrathi$ kubectl top pods
NAME                                       CPU(cores)   MEMORY(bytes)
flask-app-deployment-78fc67b66b-wrhlv   1001m        1221Mi
[(base) Romils-MacBook-Pro:test romilrathi$ kubectl top pods
NAME                                       CPU(cores)   MEMORY(bytes)
flask-app-deployment-78fc67b66b-wrhlv   1001m        1221Mi
[(base) Romils-MacBook-Pro:test romilrathi$ kubectl top pods
NAME                                       CPU(cores)   MEMORY(bytes)
flask-app-deployment-78fc67b66b-wrhlv   991m         1222Mi
(base) Romils-MacBook-Pro:test romilrathi$ 
```

The figure above shows the increase in CPU and Memory utilization to process the requests. It is scaling the resources vertically in to handle large amount of requests.

Hey Load Testing Output



```
(base) Romils-MacBook-Pro:test romilrathi$ hey -n 100000 -q 50 -z 5m -m POST -H "Content-Type: application/json" -d '{"query":"your_query_here", "topk": 10}' http://localhost:500
16/predict

Summary:
  Total:        302.7767 secs
  Slowest:      4.7075 secs
  Fastest:      0.2982 secs
  Average:      2.8112 secs
  Requests/sec: 17.7061

  Total data:   2192649 bytes
  Size/request: 409 bytes

Response time histogram:
  0.298 [1]     |
  0.739 [7]     |
  1.180 [6]     |
  1.621 [7]     |
  2.062 [5]     |
  2.503 [437]   |■■■■■
  2.944 [3762]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  3.385 [839]   |■■■■■■■■■
  3.826 [165]   |■■
  4.267 [64]    |■
  4.707 [68]    |■


Latency distribution:
  10% in 2.5107 secs
  25% in 2.6119 secs
  50% in 2.7297 secs
  75% in 2.9055 secs
  90% in 3.1860 secs
  95% in 3.4019 secs
  99% in 4.3335 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0016 secs, 0.2982 secs, 4.7075 secs
  DNS-lookup:   0.0011 secs, 0.0001 secs, 0.0371 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0018 secs
  resp wait:    2.7909 secs, 0.2632 secs, 4.7063 secs
  resp read:    0.0186 secs, 0.0000 secs, 0.1995 secs

Status code distribution:
  [200] 5361 responses
```

The test results indicated challenges in handling the request load, with throughput averaging at 17.7 requests per second (RPS). The majority of responses exhibited a slower average time of around 2.9 seconds. The response wait time is also high with 2.8.

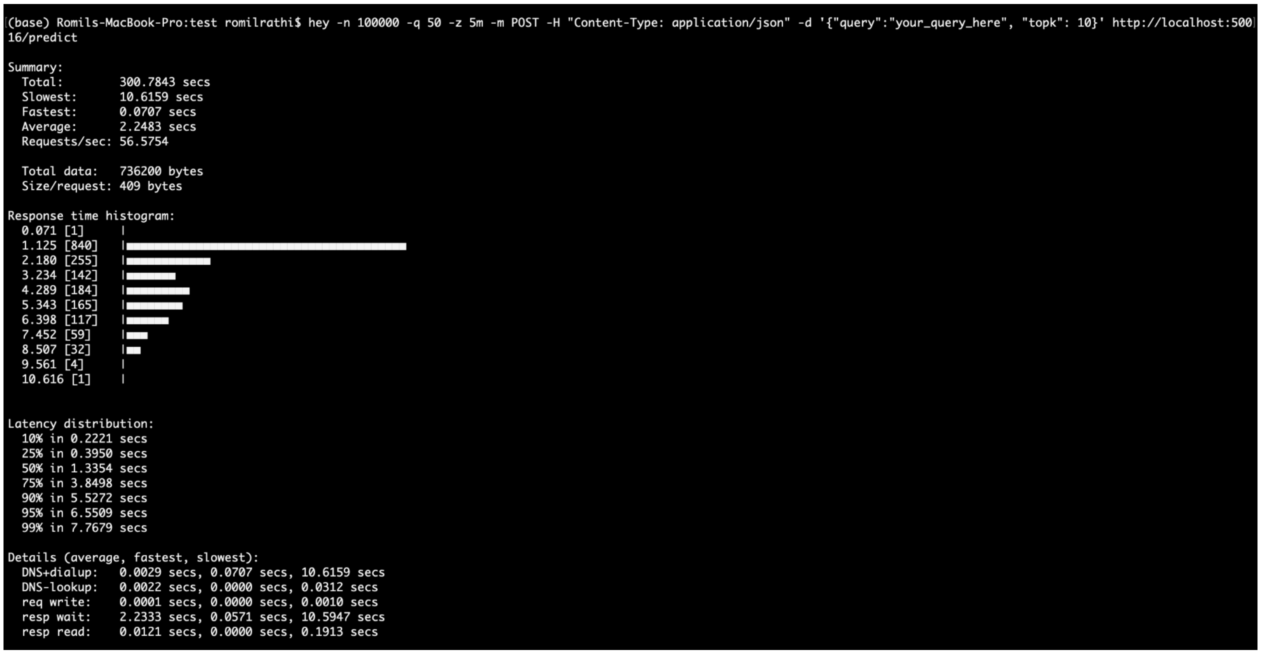
Scaling Solutions

To address performance limitations, two scaling methodologies were considered:

1**. Vertical Scaling** - Enhancing system capacity by adding more resources, such as memory or processing power, to the existing setup.
2. **Horizontal Scaling -** Increasing system capacity by adding more instances (pods) to distribute the load across multiple nodes.

Due to limitations in the local development environment regarding vertical scaling feasibility (Macbook with limited RAM), horizontal scaling was implemented by applying following command:
kubectl scale deployment flask-app-deployment --replicas=3

The results showcased significant improvements:



```
(base) Romils-MacBook-Pro:test romilrathi$ hey -n 100000 -q 50 -z 5m -m POST -H "Content-Type: application/json" -d '{"query":"your_query_here", "topk": 10}' http://localhost:500
16/predict

Summary:
  Total:        300.7843 secs
  Slowest:      10.6159 secs
  Fastest:      0.0707 secs
  Average:      2.2483 secs
  Requests/sec: 56.5754

  Total data:   736200 bytes
  Size/request: 409 bytes

Response time histogram:
  0.071 [1]     |
  1.125 [840]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  2.180 [255]   |■■■■■■■■■■■■
  3.234 [142]   |■■■■■■■
  4.289 [184]   |■■■■■■■■■
  5.343 [165]   |■■■■■■■■
  6.398 [117]   |■■■■■■
  7.452 [59]    |■■■
  8.507 [32]    |■■
  9.561 [4]     |
  10.616 [1]    |


Latency distribution:
  10% in 0.2221 secs
  25% in 0.3950 secs
  50% in 1.3354 secs
  75% in 3.8498 secs
  90% in 5.5272 secs
  95% in 6.5509 secs
  99% in 7.7679 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0029 secs, 0.0707 secs, 10.6159 secs
  DNS-lookup:   0.0022 secs, 0.0000 secs, 0.0312 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0010 secs
  resp wait:    2.2333 secs, 0.0571 secs, 10.5947 secs
  resp read:    0.0121 secs, 0.0000 secs, 0.1913 secs
```

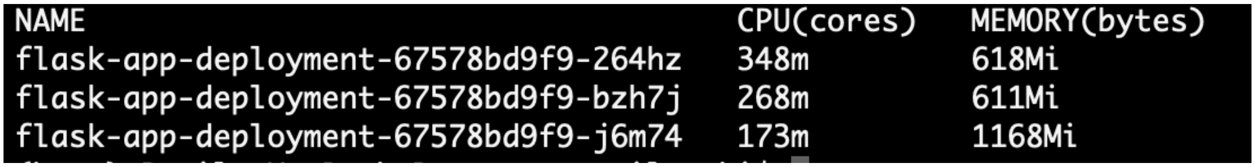
- The response time was majorly in 1.125 range as seen from the histogram above with 50% Latency distribution in under 1.3 seconds.
- Average Response Time: Reduced to 2.24 seconds.
- Throughput: Increased to 56.57, demonstrating notable performance enhancements compared to the initial test without horizontal scaling.

Test for 100 QPS

The Load Balancer Test was repeated for 100 QPS, following similar procedures as the 50 QPS test.

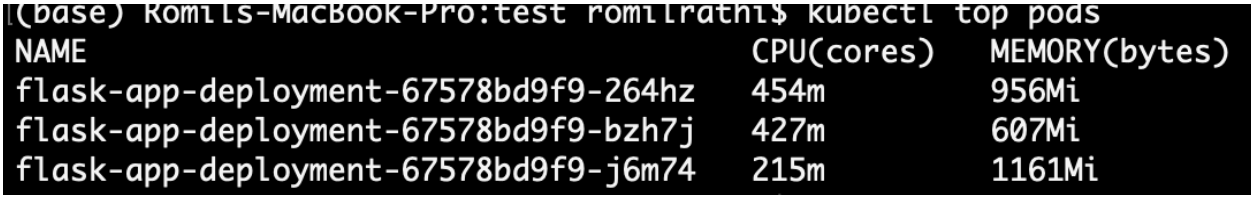
CPU Utilization Observations

The initial utilization of cores and memory is shown below:



```
NAME                                  CPU(cores)   MEMORY(bytes)
flask-app-deployment-67578bd9f9-264hz   348m         618Mi
flask-app-deployment-67578bd9f9-bzh7j   268m         611Mi
flask-app-deployment-67578bd9f9-j6m74   173m         1168Mi
```

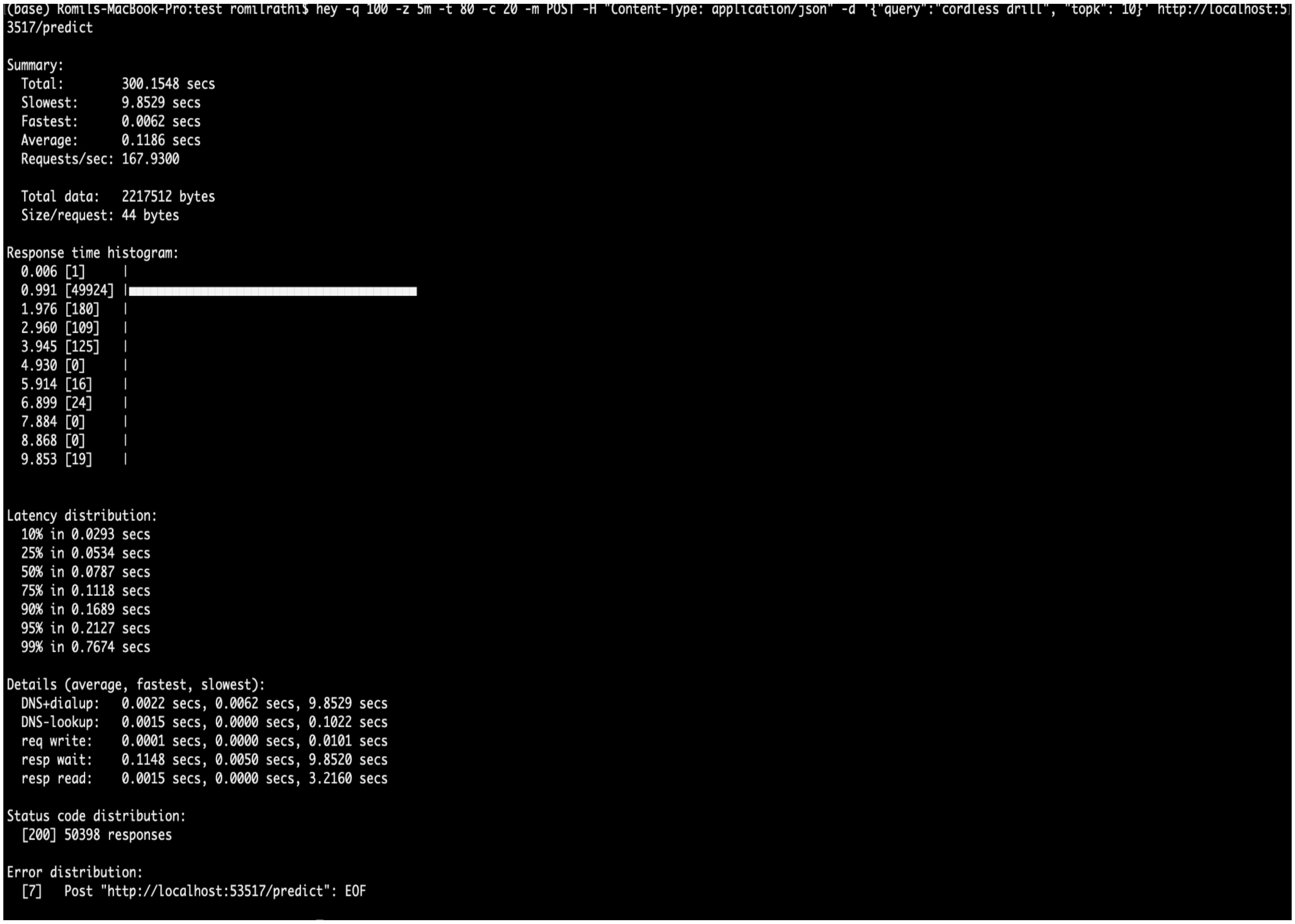
As the number of requests increases, the load balancing is able to scale the memory and cpu cores require, therefore improving the overall utilization to handle the processes.



```
(base) Romils-MacBook-Pro:test romilrathi$ kubectl top pods
NAME                                    CPU(cores)   MEMORY(bytes)
flask-app-deployment-67578bd9f9-264hz   454m         956Mi
flask-app-deployment-67578bd9f9-bzh7j   427m         607Mi
flask-app-deployment-67578bd9f9-j6m74   215m         1161Mi
```

The metrics cab ne further granualized by utilizing metrics such as % of cpu utilized etc.

Hey Load Test - 5min

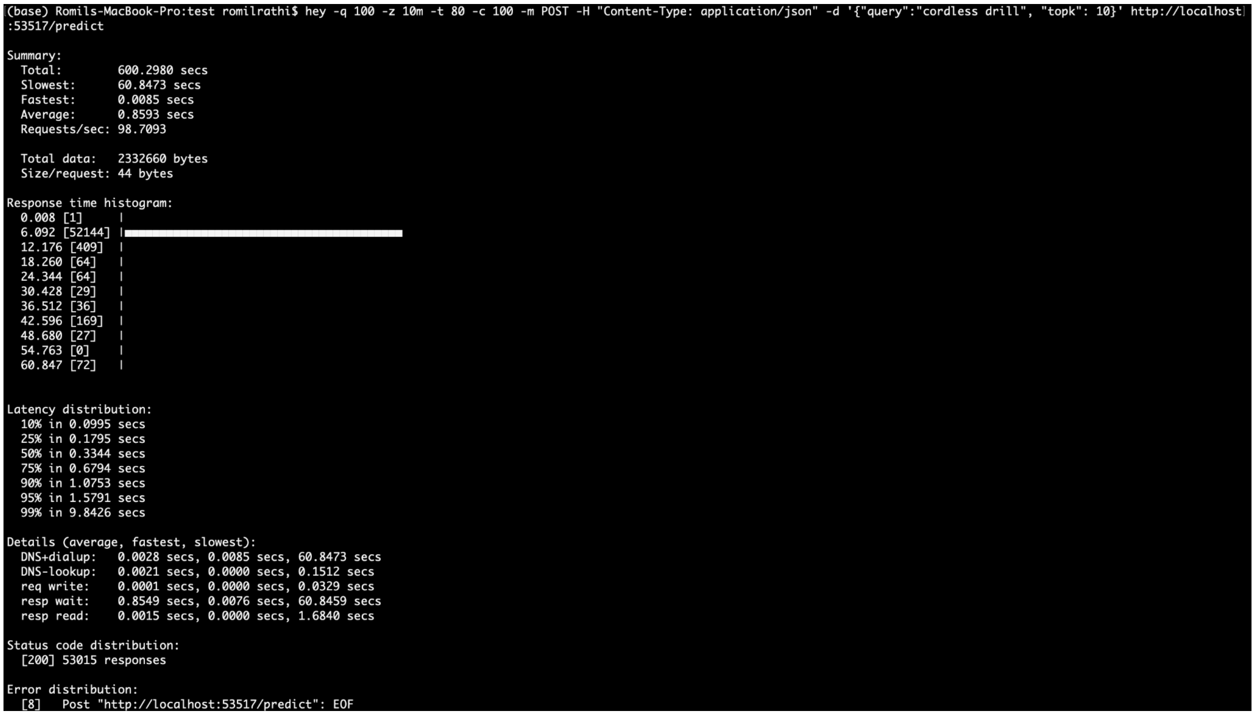


```
(base) Romils-MacBook-Pro:test romilrathi$ hey -q 100 -z 5m -t 80 -c 20 -m POST -H "Content-Type: application/json" -d '{"query":"cordless drill", "topk": 10}' http://localhost:5
3517/predict

Summary:
  Total:        300.1548 secs
  Slowest:      9.8529 secs
  Fastest:      0.0062 secs
  Average:      0.1186 secs
  Requests/sec: 167.9300

  Total data:   2217512 bytes
  Size/request: 44 bytes

Response time histogram:
  0.006 [1]     |
  0.991 [49924] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  1.976 [180]   |
  2.960 [109]   |
  3.945 [125]   |
  4.930 [0]     |
  5.914 [16]    |
  6.899 [24]    |
  7.884 [0]     |
  8.868 [0]     |
  9.853 [19]    |


Latency distribution:
  10% in 0.0293 secs
  25% in 0.0534 secs
  50% in 0.0787 secs
  75% in 0.1118 secs
  90% in 0.1689 secs
  95% in 0.2127 secs
  99% in 0.7674 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0022 secs, 0.0062 secs, 9.8529 secs
  DNS-lookup:   0.0015 secs, 0.0000 secs, 0.1022 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0101 secs
  resp wait:    0.1148 secs, 0.0050 secs, 9.8520 secs
  resp read:    0.0015 secs, 0.0000 secs, 3.2160 secs

Status code distribution:
  [200] 50398 responses

Error distribution:
  [7]   Post "http://localhost:53517/predict": EOF
```

The image above displays the result for load testing for 100 query per second using 3 replica. The test for 50 QPS shows that 1 replica is not enough which is why it was eliminated for this test. The results are as follows:

The majority of the request are fulfilled in 0.991 as observed from the histogram. The average time is 0.1186 seconds and throughput (requests/sec) are 167.93 for the 5 minute test which shows the server is performing well initially. The slowest request takes 9 seconds which shows that there is scope for improvement. Overall, 100 query per second is also resulting into error as shown from the error distribution. It can be because the server is not capable of handling such requests for long time which is why it can be further scaled vertically.

Hey Load Test - 10min



```
(base) Romils-MacBook-Pro:test romilrathi$ hey -q 100 -z 10m -t 80 -c 100 -m POST -H "Content-Type: application/json" -d '{"query":"cordless drill", "topk": 10}' http://localhost
:53517/predict

Summary:
  Total:        600.2980 secs
  Slowest:      60.8473 secs
  Fastest:      0.0085 secs
  Average:      0.8593 secs
  Requests/sec: 98.7093

  Total data:   2332660 bytes
  Size/request: 44 bytes

Response time histogram:
  0.008 [1]     |
  6.092 [52144] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  12.176 [409]  |
  18.260 [64]   |
  24.344 [64]   |
  30.428 [29]   |
  36.512 [36]   |
  42.596 [169]  |
  48.680 [27]   |
  54.763 [0]    |
  60.847 [72]   |


Latency distribution:
  10% in 0.0995 secs
  25% in 0.1795 secs
  50% in 0.3344 secs
  75% in 0.6794 secs
  90% in 1.0753 secs
  95% in 1.5791 secs
  99% in 9.8426 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0028 secs, 0.0085 secs, 60.8473 secs
  DNS-lookup:   0.0021 secs, 0.0000 secs, 0.1512 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0329 secs
  resp wait:    0.8549 secs, 0.0076 secs, 60.8459 secs
  resp read:    0.0015 secs, 0.0000 secs, 1.6840 secs

Status code distribution:
  [200] 53015 responses

Error distribution:
  [8]   Post "http://localhost:53517/predict": EOF
```

The results above are load testing performed for 10 minutes using same metrics as above for 100 query per second with 100 concurrent users.. It shows the throughput to be 98 with increase in error through the server. It can be improved by further scaling the cluster both vertically and horizontally to handle large throughput.

Further Performance Enhancement

To further improve processing capabilities beyond scaling methods, consider the following strategies:

1. Caching Mechanisms: Implement caching mechanisms to store frequently accessed data, reducing response time for subsequent requests.
2. Optimized Algorithms: Analyze and optimize algorithms used in the application to reduce computational complexity and enhance processing efficiency.
3. Load Balancing Strategies: Explore advanced load balancing strategies to intelligently distribute traffic, optimizing resource utilization across multiple nodes.
4. Run testing through other tools such as locust to see effect of concurrency.
4. Tracking Metrics with Prometheus and Grafana
Implementing Prometheus and Grafana for tracking metrics allows comprehensive monitoring of system performance. Collecting and visualizing key metrics can aid in identifying bottlenecks and optimizing resource utilization for better performance.